Symbolic Heuristic Search for Factored Markov Decision Processes

Zhengzhu Feng
Computer Science Department
University of Massachusetts
Amherst, MA 01003
fengzz@cs.umass.edu

Eric A. Hansen
Computer Science Department
Mississippi State University
Mississippi State, MS 39762
hansen@cs.msstate.edu

Abstract

We describe a planning algorithm that integrates two approaches to solving Markov decision processes with large state spaces. State abstraction is used to avoid evaluating states individually. Forward search from a start state, guided by an admissible heuristic, is used to avoid evaluating all states. We combine these two approaches in a novel way that exploits symbolic model-checking techniques and demonstrates their usefulness for decision-theoretic planning.

Introduction

Markov decision processes (MDPs) have been adopted as a framework for research in decision-theoretic planning. Classic dynamic programming algorithms solve MDPs in time polynomial in the size of the state space. However, the size of the state space grows exponentially with the number of features describing the problem. This "state explosion" problem limits use of the MDP framework, and overcoming it has become an important topic of research.

Over the past several years, approaches to solving MDPs that do not rely on complete state enumeration have been developed. One approach exploits a feature-based (or factored) representation of an MDP to create state abstractions that allow the problem to be represented and solved more efficiently (Dearden & Boutilier 1997; Hoey et al. 1999; and many others). Another approach limits computation to states that are reachable from the starting state(s) of the MDP (Barto, Bradtke, & Singh 1995; Dean *et al.* 1995; Hansen & Zilberstein 2001). In this paper, we show how to combine these two approaches. Moreover, we do so in a way that demonstrates the usefulness of symbolic model-checking techniques for decision-theoretic planning.

There is currently great interest in using symbolic model-checking to solve artificial intelligence planning problems with large state spaces. This interest is based on recognition that the problem of finding a plan (i.e., a sequence of actions and states leading from an initial state to a goal state) can be treated as a reachability problem in model checking. The planning problem is solved symbolically in the sense that reachability analysis is performed by manipulating sets of states, rather than individual states, alleviating the state explosion problem. Recent work has shown that symbolic model checking provides a very effective approach to nondeterministic planning (Cimatti, Roveri, & Traverso 1998). Nondeterministic planning is similar to decision-theoretic planning in that it considers actions with multiple outcomes, allows plan execution to include conditional and iterative behavior, and represents a plan as a mapping from states to actions. However, decision-theoretic planning is more complex than nondeterminsitic planning because it associates probabilities and rewards with state transitions. The problem is not simply to construct a plan that reaches the goal, but to find a plan that maximizes expected value.

The first use of symbolic model checking for decision-theoretic planning is the SPUDD planner (Hoey *et al.* 1999), which solves factored MDPs using dynamic programming. Although it does not use reachability analysis, it uses symbolic model-checking techniques to perform dynamic programming efficiently on sets of states. Our paper builds heavily on this work, but extends it in an important way. Whereas dynamic programming solves an MDP for all possible starting states, we use knowledge of the starting state(s) to limit planning to reachable states. In essence, we show how to improve the efficiency of the SPUDD framework by using symbolic reachability analysis together with dynamic programming. Our algorithm can also be viewed as a generalization of heuristic-search planners for MDPs, in which our contribution is to show how to perform heuristic search symbolically for factored MDPs. The advantage of heuristic search over dynamic programming is that it focuses planning resources on the relevant parts of the state space.

Factored MDPs and decision diagrams

A Markov decision process (MDP) is defined as a tuple (S, A, P, R) where: S is a set of states; A is a set of actions; P is a set of transition models $P^a : S \times S \rightarrow [0, 1]$, one for each action, specifying the transition probabilities of the process; and R is a set of reward models $R^a : S \rightarrow \Re$, one for each action, specifying the expected reward for taking action a in each state. We consider MDPs for which the objective is to find a policy $\pi : S \rightarrow A$ that maximizes total discounted reward over an infinite (or indefinite) horizon, where $\gamma \in [0, 1]$ is the discount factor. (We allow a discount factor of 1 for indefinite-horizon problems only, that is, for MDPs that terminate after a goal state is reached.)

In a factored MDP, the set of states is described by a set of random variables $\mathbf{X} = \{X_1, \ldots, X_n\}$. Without loss of generality, we assume these are Boolean variables. A particular instantiation of the variables corresponds to a unique state. Because the set of states $S = 2^{\mathbf{X}}$ grows exponentially with the number of variables, it is impractical to represent the transition and reward models explicitly as matrices when the number of states variables is large. Instead we follow Hoey *et al.*(1999) in using algebraic decision diagrams to achieve a more compact representation.

Algebraic decision diagrams (ADDs) are a generalization of binary decision diagrams (BDDs), a compact data structure for Boolean functions used in symbolic model checking. A decision diagram is a data structure (corresponding to an acyclic directed graph) that compactly represents a mapping from a set of Boolean state variables to a set of values. A BDD represents a mapping to the values 0 or 1. An ADD represents a mapping to any finite set of values. To represent these mappings compactly, decision diagrams exploit the fact that many instantiations of the state variables map to the same value. In other words, decision diagrams exploit state abstraction. BDDs are typically used to represent the characteristic functions of sets of states and the transition functions of finite-state automata. ADDs can represent weighted finite-state automata, where the weights correspond to transition probabilities or rewards, and thus are an ideal representation for MDPs.

Hoey *et al.* (1999) describe how to represent the transition and reward models of a factored MDP compactly using ADDs. We adopt their notation and refer to their paper for details of this representation. Let $\mathbf{X} = \{X_1, \ldots, X_n\}$ represent the state variables at the current time and let $\mathbf{X}' = \{X'_1, \ldots, X'_n\}$ represent the state variables at the next step. For each action, an ADD $P^a(\mathbf{X}, \mathbf{X}')$ represents the transition probabilities for the action. Similarly, the reward model $R^a(\mathbf{X})$ for each action a is represented by an ADD. The advantage of using ADDs to represent mappings from states (and state transitions) to values is that the complexity of operators on ADDs depends on the number of nodes in the diagrams, not the size of the state space. If there is sufficient regularity in the model, ADDs can be very compact, allowing problems with large state spaces to be represented and solved efficiently.

Symbolic LAO* algorithm

To solve factored MDPs, we describe a symbolic generalization of the LAO* algorithm (Hansen & Zilberstein 2001). LAO* is an extension of the classic search algorithm AO* that can find solutions with loops. This makes it possible for LAO* to solve MDPs, since a policy for an infinite-horizon MDP allows both conditional and cyclic behavior. Like AO*, LAO* has two alternating phases. First, it expands the best partial solution (or policy) and evaluates the states on its fringe using an admissible heuristic function. Then it performs dynamic programming on the states visited by the best partial solution, to update their values and possibly revise the currently best partial solution. The two phases alternate until a complete solution is found, which is guaranteed to be optimal.

AO* and LAO* differ in the algorithms they use in the dynamic programming step. Because AO* assumes an acyclic solution, it can perform dynamic programming in a single backward pass from the states on the fringe of the solution to the start state. Because LAO* allows solutions with cycles, it relies on an iterative dynamic programming algorithm (such as value iteration or policy iteration). In organization, the LAO* algorithm is similar to the "envelope" dynamic programming approach to solving MDPs (Dean *et al.* 1995). It is also closely related to RTDP (Barto, Bradtke, & Singh 1995), which is an on-line (or "real time") search algorithm for MDPs, in contrast to LAO*, which is an off-line search algorithm.

We call our generalization of LAO* a symbolic search algorithm because it manipulates sets of states, instead of individual states. In keeping with the symbolic model-checking approach, we represent a set of states S by its characteristic function χ_S, so that $s \in S \iff \chi_S(s) = 1$. We represent the characteristic function of a set of states by an ADD. (Because its values are 0 or 1, we can also represent a characteristic function by a BDD.) From now on, whenever we refer to a set of states, S, we implicitly refer to its characteristic function, as represented by a decision diagram.

In addition to representing sets of states as ADDs, we represent every element manipulated by the LAO* algorithm as an ADD, including: the transition and reward models; the policy $\pi : S \rightarrow A$; the state evaluation function $V : S \rightarrow \Re$ that is computed in the course of finding a policy; and an admissible heuristic evaluation function $h : S \rightarrow \Re$ that guides the search for the best policy. Even the discount factor γ is represented by a simple ADD that maps every input to a constant value. This allows us to perform all computations of the LAO* algorithm using ADDs.

Besides exploiting state abstraction, we want to limit computation to the set of states that are reachable from the start state by following the best policy. Although an ADD effectively assigns a value to every state, these values are only relevant for the set of reachable states. To focus computation on the relevant states, we introduce the notion of *masking* an ADD. Given an ADD D and a set of relevant states U, masking is performed by multiplying D by χ_U. This has the effect of mapping all irrelevant states to the value zero. We let D_U denote the resulting *masked ADD*. (Note that we need to have U in order to correctly interpret D_U). Mapping all irrelevant states to zero can simplify the ADD considerably. If the set of reachable states is small, the masked ADD often has dramatically fewer nodes. This in turn can dramatically improve the efficiency of computation using ADDs.[1]

Our symbolic implementation of LAO* does not maintain an explicit search graph. It is sufficient to keep track of the set of states that have been "expanded" so far, denoted G, the *partial value function*, denoted V_G, and a *partial policy*,

[1] Although we map the values of irrelevant states to zero, it does not matter what value they have. This suggests a way to simplify a masked ADD further. After mapping irrelevant states to zero, we can change the value of a irrelevant state to any other non-zero value whenever doing so further simplifies the ADD.

```
policyExpansion($\pi, S^0, G$)
1.  $E = F = \emptyset$
2.  $from = S^0$
3.  REPEAT
4.    $to = \bigcup_a Image(from \cap S^a_\pi, P^a)$
5.    $F = F \cup (to - G)$
6.    $E = E \cup from$
7.    $from = to \cap G - E$
8.  UNTIL ($from = \emptyset$)
9.  $E = E \cup F$
10. $G = G \cup F$
11. RETURN $(E, F, G)$

valueIteration($E, V$)
12. $saveV = V$
13. $E' = \bigcup_a Image(E, P^a)$
14. REPEAT
15.   $V' = V$
16.   FOR each action $a$
17.       $V^a = R^a_E + \gamma \sum_{E'} P^a_{E \cup E'} V'_{E'}$
18.   $M = \max_a V^a$
19.   $V = M_E + saveV_{\overline{E}}$
20.   residual $= \|V_E - V'_E\|$
21. UNTIL stopping criterion met
22. $\pi = extractPolicy(M, \{V^a\})$
23. RETURN $(V, \pi, residual)$

LAO*($\{P^a\}, \{R^a\}, \gamma, S^0, h, threshold$)
24. $V = h$
25. $G = \emptyset$
26. $\pi = 0$
27. REPEAT
28.     $(E, F, G) = policyExpansion(\pi, S^0, G)$
29.     $(V, \pi, residual) = valueIteration(E, V)$
30. UNTIL ($F = \emptyset$) AND (residual $\leq$ threshold)
31. RETURN $(\pi, V, E, G)$.
```

Table 1: Symbolic LAO* algorithm.

denoted π_G. For any state in G, we can "query" the policy to determine its associated action, and compute its successor states. Thus, the graph structure is implicit in this representation. Note that throughout the whole LAO* algorithm, we only maintain one value function V and one policy π. V_G and π_G are implicitly defined by G and the masking operation.

Symbolic LAO* is summarized in Table 1. In the following, we give a more detailed explanation.

Policy expansion In the policy expansion step of the algorithm, we perform reachability analysis to find the set of states F that are not in G (i.e., have not been "expanded" yet), but are reachable from the set of start states, S^0, by following the partial policy π_G. These states are on the "fringe" of the states visited by the best policy. We add them to G and to the set of states $E \subseteq G$ that are visited by the current partial policy. This is analogous to "expanding" states on the frontier of a search graph in heuristic search. Expanding a partial policy means that it will be defined for a larger set of states in the dynamic-programming step.

Symbolic reachability analysis using decision diagrams is widely used in VLSI design and verification. Our policy-expansion algorithm is similar to the traversal algorithms used for sequential verification, but is adapted to handle the more complex system dynamics of an MDP. The key operation in reachability analysis is computation of the *image* of a set of states, given a transition function. The image is the set of all possible successor states. To perform this operation, it is convenient to convert the ADD $P^a(\mathbf{X}, X')$ to a BDD $T^a(\mathbf{X}, X')$ that maps state transitions to a value of one if the transition has a non-zero probability, and otherwise zero. The image computation is faster using BDDs than ADDs. Mathematically, the image is computed using the relational-product operator, defined as follows:

$$Image_{\mathbf{X}'}(S, T^a) = \exists \mathbf{x} \left[T^a(\mathbf{X}, X') \wedge \chi_S(\mathbf{X}) \right].$$

The conjunction $T^a(\mathbf{X}, X') \wedge \chi_S(\mathbf{X})$ selects the set of valid transitions and the existential quantification extracts and unions the successor states together. Both the relational-product operator and symbolic traversal algorithms are well studied in the symbolic model checking literature, and we refer to that literature for details about how this is computed, for example, (Somenzi 1999).

The *image* operator returns a characteristic function over $\mathbf{X}'$ that represents the set of reachable states *after* an action is taken. The assignment in line 4 implicitly converts this characteristic function so that it is defined over $\mathbf{X}$, and represents the current set of states ready for the next expansion.

Because a policy is associated with a set of transition functions, one for each action, we need to invoke the appropriate transition function for each action when computing successor states under a policy. For this, it is useful to represent the partial policy π_G in another way. We associate with each action a the set of states for which the action to take is a under the current policy, and call this set of states S^a_π. Note that $S^a_\pi \cap S^{a'}_\pi = \emptyset$ for $a \neq a'$, and $\cup_a S^a_\pi = G$. Given this alternative representation of the policy, line 4 computes the set of successor states following the current policy using the *image* operator.

Dynamic programming The dynamic-programming step of LAO* is performed using a modified version of the SPUDD algorithm. The original SPUDD algorithm performs dynamic programming over the entire state space. We modify it to focus computation on reachable states, using the idea of masking. Masking lets us perform dynamic programming on a subset of the state space instead of the entire state space. The pseudocode in Table 1 assumes that dynamic programming is performed on E, the states visited by the currently best (partial) policy. This has been shown to lead to the best performance of LAO*, although a larger or smaller set of states can also be updated (Hansen & Zilberstein 2001). Note that all ADDs used in the dynamic-programming computation are masked to improve efficiency.

Because π_G is a partial policy, there can be states in E with successor states that are not in G, denoted E'. This is true until LAO* converges. In line 13, we identify these states so that we can do appropriate masking. To perform dynamic programming on the states in E, we assign admissible values to the "fringe" states in E', where these values come from the current value function. Note that the value function is initialized to an admissible heuristic evaluation function at the beginning of the algorithm.

With all components properly masked, we can perform dynamic programming using the SPUDD algorithm. This is summarized in line 17. The full equation is

$$V^a(\mathrm{X}) = R^a_E(\mathrm{X}) + \gamma \sum_{E'} P^a_{E \cup E'}(\mathrm{X}, X') \cdot V'_{E'}(\mathrm{X}').$$

The masked ADDs R^a_E and $P^a_{E \cup E'}$ need to be computed only once for each call to $valueIteration()$ since they don't change between iterations. Note that the product $P^a_{E \cup E'} \cdot V'_{E'}$ is effectively defined over $E \cup E'$. After the summation over E', which is accomplished by existentially abstracting away all post-action variables, the resulting ADD is effectively defined over E only. As a result, V^a is effectively a masked ADD over E, and the maximum M at line 18 is also a masked ADD over E. In line 19, we use the notation M_E to emphasize that V is set equal to the newly computed values for E and the saved values for the rest of the state space. There is no masking computation performed.

The residual in line 20 can be computed by finding the largest absolute value of the ADD $(V_E - V'_E)$. We use the masking subscript here to emphasize that the residual is computed only for states in the set E. The masking operation can actually be avoided here since at this step, $V_E = M$, which is computed in line 18, and V'_E is the M from the previous iteration.

Dynamic programming is the most expensive step of LAO*, and it is usually not efficient to run it until convergence each time this step is performed. Often a single iteration gives the best performance. After performing value iteration, we extract a policy in line 22 by comparing M against the action value function V^a (breaking ties arbitrarily):

$$\forall s \in E \;\; \pi(s) = a \;\text{ if }\; M(s) = V^a(s).$$

The symbolic LAO* algorithm returns a value function V and a policy π, together with the set of states E that are visited by the policy, and the set of states G that have been "expanded" by LAO*.

Convergence test At the beginning of LAO*, the value function V is initialized to the admissible heuristic h that overestimates the optimal value function. Each time value iteration is performed, it starts with the current values of V. Hansen and Zilberstein (2001) show that these values decrease monotonically in the course of the algorithm; are always admissible; and converge arbitrarily close to optimal. LAO* converges to an optimal or ϵ-optimal policy when two conditions are met: (1) its current policy does not have any unexpanded states, and (2) the error bound of the policy is less than some predetermined threshold. Like other heuristic search algorithms, LAO* can find an optimal solution without visiting the entire state space. The convergence proofs for the original LAO* algorithm carry over in a straightforward way to symbolic LAO*.

Admissible heuristics LAO* uses an admissible heuristic to guide the search. Because a heuristic is typically defined for all states, a simple way to create an admissible heuristic is to use dynamic programming to create an approximate value function. Given an error bound on the approximation, the value function can be converted to an admissible heuristic. (Another way to ensure admissibility is to perform value iteration on an initial value function that is admissible, since each step of value iteration preserves admissibility.) Symbolic dynamic programming can be used to compute an approximate value function efficiently. St. Aubin et al. (2000) describe an approximate dynamic programming algorithm for factored MDPs, called APRICODD, that is based on SPUDD. It simplifies the value function ADD by aggregating states with similar values. Another approach to approximate dynamic programming for factored MDPs described by Dearden and Boutilier (1997) can also be used to compute admissible heuristics.

Use of dynamic programming to compute an admissible heuristic points to a two-fold approach to solving factored MDPs. First, dynamic programming is used to compute an approximate solution for all states that serves as a heuristic. Then heuristic search is used to find an exact solution for a subset of reachable states.

Experimental results

Table 2 compares the performance of LAO* and SPUDD on the factory examples (f to f6) used by Hoey *et al.*(1999) to test the performance of SPUDD, as well as some additional examples (a1 to a4). We use additional test examples because many of the state variables in the factory examples represent resources that cannot be affected by any action. As a result, we found that only a small number of states are reachable from a given start state in these examples. Examples a1 to a4 (which are artificial examples) are structured so that every state variable can be changed by some action, and thus, most or all of the state space can be reached from any start state. Such examples present a greater challenge to a heuristic-search approach.

Because the performance of LAO* depends on the start state, the experimental results reported for LAO* in Table 2 are averages for 50 random starting states. To create an admissible heuristic, we performed several iterations (ten for the factory examples and twenty for the others) of an approximate value iteration algorithm similar to APRICODD (St-Aubin, Hoey, & Boutilier 2000). The algorithm was started with an admissible value function created by assuming the maximum reward is received each step. The time used to compute the heuristic for these examples is between 2% and 8% of the running time of SPUDD on the same examples. Experiments were performed on a Sun UltraSPARC II with a 300MHz processor and 2 gigabytes of memory.

LAO* achieves its efficiency by focusing computation on a subset of reachable states. The column labelled $|E|$ shows the average number of states visited by an optimal policy, beginning from a random start state. Clearly, the factory examples have an unusual structure, since an optimal policy for these examples visits very few states. The numbers are

Example			Reachability Results			Size Results				Timing Results				
			Symb-LAO*			Symb-LAO*		SPUDD		Symb-LAO*			LAO*	SPUDD
	$\lvert S\rvert$	$\lvert A\rvert$	$\lvert E\rvert$	$\lvert G\rvert$	reach	N	L	N	L	exp.	DP	total	total	total
f	2^{17}	14	5	105	190	55	5	1220	246	0.3	6.4	6.7	3.8	34.5
f0	2^{19}	14	5	62	132	61	5	1597	246	0.2	3.5	3.7	3.7	46.2
f1	2^{21}	14	4	54	107	54	4	3101	327	0.1	3.6	3.7	3.7	101.6
f2	2^{22}	14	4	66	125	53	4	3101	327	0.2	4.1	4.4	4.1	105.0
f3	2^{25}	15	4	59	136	74	4	9215	357	0.2	4.7	4.9	3.7	289.1
f4	2^{28}	15	4	49	125	78	4	22170	527	0.1	5.0	5.2	3.7	645.3
f5	2^{31}	18	5	218	509	83	4	44869	1515	1.2	35.9	37.3	4.4	1524.2
f6	2^{35}	23	9	1419	2386	106	5	169207	3992	13.5	771.5	792.6	9.3	7479.5
a1	2^{20}	25	3.0×10^{3}	3.3×10^{3}	1.0×10^{6}	181	19	15758	4056	0.5	53.6	54.0	39.01	12774.3
a2	2^{20}	30	1.0×10^{4}	3.3×10^{4}	1.0×10^{6}	6240	2190	9902	4594	57.7	1678.5	1738.1	4581.9	10891.7
a3	2^{30}	10	1.8×10^{6}	1.9×10^{6}	6.7×10^{7}	3522	439	25839	6434	7.3	344.8	352.1	NA	11169.9
a4	2^{40}	10	9.6×10^{4}	2.8×10^{6}	1.7×10^{10}	99	4	NA	NA	0.7	75.5	76.3	NA	NA

Table 2: Performance comparison of LAO* (both symbolic and non-symbolic) and SPUDD.

much larger for examples a1 through a4. The column labeled *reach* shows the average number of states that can be reached from the start state, by following *any* policy. The column labelled $|G|$ is important because it shows the number of states "expanded" by LAO*. These are states for which a backup is performed at some point in the algorithm, and this number depends on the quality of the heuristic. The better the heuristic, the fewer states need to be expanded to find an optimal policy. The gap between $|E|$ and *reach* reflects the potential for increased efficiency using heuristic search, instead of simple reachability analysis. The gap between $|G|$ and *reach* reflects the actual increased efficiency.

The columns labeled N and L, under LAO* and SPUDD respectively, compare the size of the final value function returned by symbolic LAO* and SPUDD. The columns under N give the number of nodes in the respective value function ADDs, and the columns under L give the number of leaves. Because LAO* focuses computation on a subset of the state space, it finds a much more compact solution (which translates into increased efficiency).

The last five columns compare the average running times of symbolic LAO* to the running times of non-symbolic LAO* and SPUDD. Times are given in CPU seconds. For many of these examples, the MDP model is too large to represent explicitly. Therefore, our implementation of non-symbolic LAO* uses the same ADD representation of the MDP model as symbolic LAO* and SPUDD. However, non-symbolic LAO* performs heuristic search in the conventional way by creating a search graph in which the nodes correspond to "flat" states that are enumerated individually.

The total running time of symbolic LAO* is broken down into two parts; the column "exp." shows the average time for policy expansion and the column "DP" shows the average time for dynamic programming. These results show that dynamic programming consumes most of the running time. This is in keeping with a similar observation about the original (non-symbolic) LAO* algorithm. The time reported for dynamic programming includes the time for masking. For this set of examples, masking takes between 0.5% and 2.1% of the running time of the dynamic programming step. The last three columns compare the average time it takes symbolic LAO* to solve each problem, for 50 random starting states, to the running times of non-symbolic LAO* and SPUDD. This comparison leads to several observations.

First, we note that the running time of non-symbolic LAO* is correlated with $|G|$, the number of states evaluated (i.e., expanded) during the search, which in turn is affected by the starting state, the reachability structure of the problem, and the accuracy of the heuristic function. As $|G|$ increases, the running time of non-symbolic LAO* increases. The search graphs for examples a3 and a4 are so large that these problems cannot be solved using non-symbolic LAO*. (NA indicates that the problem could not be solved.)

The running time of symbolic LAO* depends not only on $|G|$, but on the degree of state abstraction the symbolic approach achieves in representing the states in G. For the factory examples and example a1, the number of states evaluated by LAO* is small enough that the overhead of symbolic search outweighs the improved efficiency from state abstraction. For these examples, symbolic LAO* is somewhat slower than non-symbolic LAO*. But for examples a2 to a4, the symbolic approach significantly – and sometimes dramatically – improves the performance of LAO*. Symbolic LAO* also outperforms SPUDD for all examples. This is to be expected since LAO* solves the problem for only part of the state space. Nevertheless, it demonstrates the power of using heuristic search to focus computation on relevant states.

We conclude by noting that examples a3 and a4 are beyond the range of both SPUDD and non-symbolic LAO*, or can only be solved with great difficulty. Yet symbolic LAO* solves both examples efficiently. This illustrates the advantage of combining heuristic search and state abstraction, rather than relying on either approach alone.

Related work

As noted in the introduction, symbolic model checking techniques have been used previously for nondeterministic planning. In both nondeterministic and decision-theoretic planning, plans may contain cycles that represent iterative, or "trial-and-error," strategies. In nondeterministic planning, the concept of a *strong cyclic plan* plays a central role (Cimatti, Roveri, & Traverso 1998; Daniele, Traverso, & Vardi 1999). It refers to a plan that contains an iterative strategy and is guaranteed to eventually achieve the goal. The concept of a strong cyclic plan has an interesting analogy in decision-theoretic planning. LAO* was originally developed for the framework of stochastic shortest-path problems. A stochastic shortest-path problem is an MDP with a goal state, where the objective is to find an optimal policy (usually containing cycles) among policies that reach the goal state with probability one. A policy that reaches the goal with probability one, also called a *proper policy*, can be viewed as a probabilistic generalization of the concept of a strong cyclic plan. In this respect and others, the symbolic LAO* algorithm presented in this paper can be viewed as a decision-theoretic generalization of symbolic algorithms for nondeterministic planning.

One difference is that the algorithm presented in this paper uses heuristic search to limit the number of states for which a policy is computed. An integration of symbolic model checking with heuristic search has not yet been explored for nondeterministic planning. However, Edelkamp & Reffel(1998) describe a symbolic generalization of A* that combines symbolic model checking and heuristic search in solving deterministic planning problems. A combined approach has also been explored for conformant planning (Bertoli, Cimatti, & Roveri 2001).

In motivation, our work is closely related to the framework of *structured reachability analysis*, which exploits reachability analysis in solving factored MDPs (Boutilier, Brafman, & Geib 1998). However, there are important differences. The symbolic model-checking techniques we use differ from the approach to state abstraction used in that work, which is derived from GRAPHPLAN. More importantly, their concept of reachability analysis is weaker than the approach adopted here. In their framework, states are considered irrelevant if they cannot be reached from the start state by following *any policy*. By contrast, our approach considers states irrelevant if it can be proved (by gradually expanding a partial policy guided by an admissible heuristic) that these states cannot be reached from the start state by following *an optimal policy*. Use of an admissible heuristic to limit the search space is characteristic of heuristic search, in contrast to simple reachability analysis. As Table 2 shows, LAO* evaluates much less of the state space than simple reachability analysis. The better the heuristic, the smaller the number of states it examines.

Conclusion

We have described a symbolic generalization of LAO* that solves factored MDPs using heuristic search. Given a start state, LAO* uses an admissible heuristic to focus computation on the parts of the state space that are reachable from the start state. The stronger the heuristic, the greater the focus and the more efficient a planner based on this approach. Symbolic LAO* also exploits state abstraction using symbolic model checking techniques. It can be viewed as a decision-theoretic generalization of symbolic approaches to nondeterministic planning.

To solve very large MDPs, we believe that decision-theoretic planners will need to employ a collection of complementary strategies that exploit different forms of problem structure. Showing how to combine heuristic search with symbolic techniques for state abstraction is a step in this direction. Integrating additional strategies into a decision-theoretic planner is a topic for future research.

Acknowledgments This work was supported, in part, by NSF grant IIS-9984952 and by NASA grants NAG-2-1463 and NAG-2-1394.

References

Barto, A.; Bradtke, S.; and Singh, S. 1995. Learning to act using real-time dynamic programming. *Artificial Intelligence* 72:81–138.

Bertoli, P.; Cimatti, A.; and Roveri, M. 2001. Heuristic search + symbolic model checking = efficient conformant planning. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, 467–472.

Boutilier, C.; Brafman, R. I.; and Geib, C. 1998. Structured reachability analysis for Markov decision processes. In *Proceedings of the 14th International Conference on Uncertainty in Artificial Intelligence*, 24–32.

Cimatti, M.; Roveri, M.; and Traverso, P. 1998. Automatic OBDD-based generation of universal plans in nondeterministic domains. In *Proceedings of the 15th National Conference on Artificial Intelligence*, 875 – 881.

Daniele, M.; Traverso, P.; and Vardi, M. 1999. Strong cyclic planning revisited. In *Proceedings of the 5th European Conference on Planning*.

Dean, T.; Kaelbling, L.; Kirman, J.; and Nicholson, A. 1995. Planning under time constraints in stochastic domains. *Artificial Intelligence* 76:35–74.

Dearden, R., and Boutilier, C. 1997. Abstraction and approximate decision-theoretic planning. *Artificial Intelligence* 89:219–283.

Edelkamp, S., and Reffel, F. 1998. OBDDs in heuristic search. In *German Conference on Artificial Intelligence (KI)*, 81–92.

Hansen, E., and Zilberstein, S. 2001. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence* 129:35–62.

Hoey, J.; St-Aubin, R.; Hu, A.; and Boutilier, C. 1999. SPUDD: Stochastic planning using decision diagrams. In *Proceedings of the 15th Conference on Uncertainty in Articial Intelligence*, 279–288.

Somenzi, F. 1999. Binary decision diagrams. In Broy, M., and Steinbruggen, R., eds., *Calculational System Design*, volume 173 of *NATO Science Series F: Computer and Systems Sciences*. IOS Press. 303–366.

St-Aubin, R.; Hoey, J.; and Boutilier, C. 2000. APRICODD: Approximate policy construction using decision diagrams. In *Proceedings of NIPS-2000*.